

## 分布式系统中局部处理机的设计与实现

魏 敏\*, 刘以安, 吴鸿雁

(江南大学 数字媒体学院, 江苏 无锡 214122)

(\*通信作者电子邮箱 alvar\_wm@126.com)

**摘 要:**针对企业生产过程中存在大量原始数据需要实时处理的问题,设计并实现了一个基于自定义架构的局部处理机。在设计之初以 Hadoop 的并行架构为参考,对 MapReduce 的工作原理和缓存方式进行了分析,在此基础上根据实际生产环境设计了一个“多类线程协同处理”的程序架构,并辅以两类自定义的数据缓存方式,保证了分布式系统中的局部处理机在接收、计算、上传各环节的并发性和正确性。该系统投入实际生产并连续使用一年有余,实现了将企业多个车间生成的原始数据进行实时处理的预期目标,具有很好的稳定性、有效性和可扩展性。实际应用结果表明,自定义的程序架构和有效的缓存方式能实现大量数据的同步处理及分析。

**关键词:** Hadoop; MapReduce; 数据缓存; 并发; 局部处理机; 分布式系统

**中图分类号:** TP316.4 **文献标志码:** A

### Design and implementation of local processor in a distributed system

WEI Min\*, LIU Yi'an, WU Hongyan

(School of Digital Media, Jiangnan University, Wuxi Jiangsu 214122, China)

**Abstract:** Concerning the problem that there is a lot of data which need to be real-time processed during the production process, the local processor, based on multi-thread and co-processing architecture and two data buffer mechanisms was accomplished. As a reference, multi-functional thread in Hadoop's parallel architecture has an impressed impact on the design of the local processor, especially MapReduce principle. Based on the user-defined architecture, the local processor ensures data concurrency and correctness during receiving, computing and uploading. The system has been put into production for over one year. It can meet the enterprise requirements and has good stability, real-time, effectiveness and scalability. The application result shows that the local processor can achieve synchronized analysis and processing of mass data.

**Key words:** Hadoop; MapReduce; data buffer; concurrency; local processor; distributed system

## 0 引言

在实际应用中,很多企业基于 LabVIEW (Laboratory Virtual Instrument Engineering Workbench) 生成的数据需要存入数据库,以便进行深层次的数据分析。本文涉及的情况是若干上位机通过传输控制协议 (Transmission Control Protocol, TCP) 将 LabVIEW 采集的原始数据传送至局部处理机,再由局部处理机对各数据分类、缓存、计算、重组、上传至数据库中,供各级用户按需使用。本文应用的架构属于松耦合式分布式系统,分布式系统的开发经历<sup>[1]</sup>表明,作为分布式系统中的一个执行机构,处理机的实时处理能力对系统的整体性能有着举足轻重的影响,如果采用了不适合的并发模型,不仅会增加开发难度,甚至可能导致系统不能正常工作,所以在设计该处理机时以 Hadoop 的工作架构为参考,并对 MapReduce 的工作原理进行了分析。

Hadoop 是目前最通用的并行处理大规模数据的开源分布式计算平台<sup>[2]</sup>,其核心组件是 Hadoop 分布式文件系统 (Hadoop Distributed File System, HDFS) 和 MapReduce 编程模型<sup>[3-4]</sup>,两者均采用了 Master/Slave 结构<sup>[5]</sup>。

本文在对 Hadoop 的多种功能线程互为支持的工作架构进行详细分析的基础上,结合实际的工作环境,自定义并实现了一个“多类线程协同处理”的工作架构,以保障整个系统的实时性、并发性和准确性。

## 1 Hadoop 分布式编程框架

### 1.1 HDFS 设计说明

从组织结构上看 Hadoop 分 3 个部分: Clients、Masters 和 Slaves,如图 1 所示。Clients 指 Hadoop 中不属于主、从节点的其他集群设置;主节点由 HDFS 中的 NameNode、Secondary NameNode 和 MapReduce 中的 JobTracker 构成;从节点由实现数据存储的 DataNode 和实现指令计算的 TaskTracker 组成,其中 DataNode 归属于 HDFS, TaskTracker 归属于 MapReduce。

工作时,如图 2 所示, Clients 把数据分成若干模块 (除了最后一个模块,其他模块大小统一),然后通过 TCP 和 NameNode 通信,由 NameNode 根据 Rack Awareness 规则来制定 DataNode 列表并提供给 Clients。Rack Awareness 规则简言之就是将每块数据复制 3 份 (默认设置),两份存储在同一个机架架上,另外一份存储在另一个机架架上,这样可保证网络的通

收稿日期: 2014-12-19; 修回日期: 2015-02-07。

**作者简介:** 魏敏 (1979 -), 女, 江苏扬州人, 讲师, 硕士, 主要研究方向: 模式识别、图像处理; 刘以安 (1963 -), 男, 江苏涟水人, 教授, 博士, CCF 会员, 主要研究方向: 数据融合、雷达对抗、模式识别、智能系统; 吴鸿雁 (1961 -), 女, 上海人, 副教授, 硕士, 主要研究方向: 人机界面、数据挖掘。

信速度并降低机架故障带来的影响。Clients 得到 DataNode 列表后,通过 TCP 将每块数据直接写入 DataNode 中,具体实现方式为 3 个 DataNode 会打开一个同步通道,当一个 DataNode 接收到数据后,会通过通道把数据复制给其他 DataNode。NameNode 中只记录文件系统元数据,每个 DataNode 定期向 NameNode 发送心跳包和该 DataNode 包含的所有数据块列表,NameNode 则通过响应心跳来控制 DataNode,并完成节点位置信息更新等工作。如果 NameNode 死亡,Secondary NameNode 保留的文件可用于恢复 NameNode。

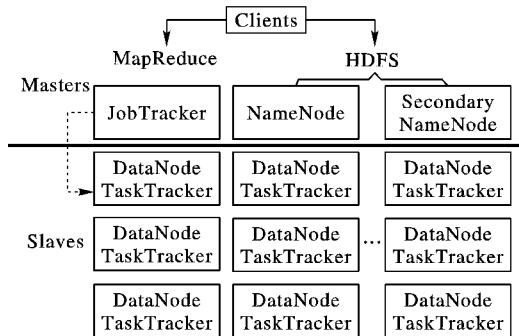


图1 Hadoop 结构示意图

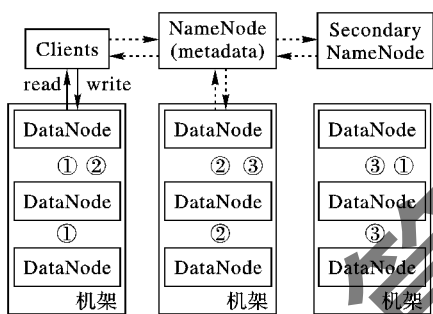


图2 HDFS 说明示意图

## 1.2 MapReduce 设计说明

部署好数据后,就由 MapReduce 完成 Clients 提交的计算作业。如图 3 所示,在 Hadoop 中只有一个 JobTracker 进程,而 TaskTracker 进程有若干个,运行在 DataNode 上。在具体工作时,每个 TaskTracker 拥有若干可并行执行任务的 Slot 槽,这些 Slot 槽分为执行 Map 任务的 Map Slot 和执行 Reduce 任务的 Reduce Slot。工作时,Clients 向 JobTracker 提交了计算作业,JobTracker 与 NameNode 通信获取数据所在的 DataNode 列表,如果 DataNode 对应的 TaskTracker 运行任务过多时,TaskTracker 也是通过“心跳”程序与 JobTracker 保持通信,所以可将该信息反馈给 JobTracker,JobTracker 再通过 NameNode 的 Rack Awareness 机制将数据传递给其他 DataNode 进行处理。至此通过 JobTracker 的宏观调控,Clients 的计算作业被分配到不同的 DataNode 上去执行,TaskTracker 在分配的 DataNode 上执行 Map 或 Reduce 任务,完成具体的计算处理工作。

MapReduce 在工作时,Map 任务是计算处理的第一步,每一个 Map 任务的输出会生成很多的中间文件,这些文件写入 Map 任务的循环内存缓冲区,当数据量达到阈值时,就启动一个后台线程把缓冲区中的内容存到磁盘。通过这种本地缓存的方式,一方面避免出现网络传输量增大的情况,另一方面避免出现频繁的磁盘操作导致性能下降的情况。当 Map 任务

完成之后,TaskTracker 通过心跳程序通知 JobTracker,JobTracker 将 Map 输出文件的地址告知 Reduce,输出文件会作为 Reduce 任务的输入数据拷贝给相对应的 Reduce。Reduce 任务把若干 Map 的输出数据做筛选、合并、统计等工作,并输出最终结果。

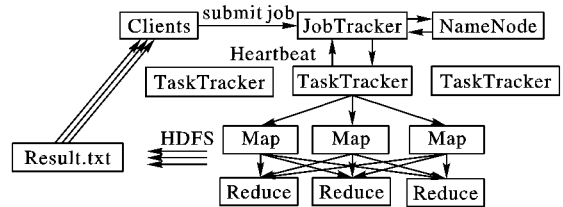


图3 MapReduce 说明示意图

通过对 Hadoop 的结构分析,可以了解到其无论是以块序列的形式存储数据资源的过程中,还是以并行处理框架对数据进行计算的过程中,Hadoop 定义的线程具有功能多样性的特征,在事务处理过程中不同功能的线程各司其职,前台后台线程互为条件,并且采用缓存的方式妥善处理计算的中间结果,由此可见一个合理而又科学的工作架构对于系统的运行起着举足轻重的作用。本文实现的处理机受此启发,自定义并实现了一个多类线程并发协作的工作框架,并配合两种自定义的缓存方式,保证系统的准确运行。

## 2 局部处理机

### 2.1 工作模型及说明

本文实现的局部处理机处于整个分布式系统的一个中间环节,主要工作可归纳为:1) 端口监听,按协议格式实时接收多路生产数据;2) 数据处理,接收到的数据在本地同步进行分类、重组;3) 数据上传,处理后的数据上传至终端的数据库中。处理机接收到的数据根据使用需求可分为两大类:设备工况类数据和文档类数据。设备工况类数据要求能在终端对设备的整个工作过程进行回放;文档类数据要求能在终端对生产的原始数据进行阅读、统计、分析。

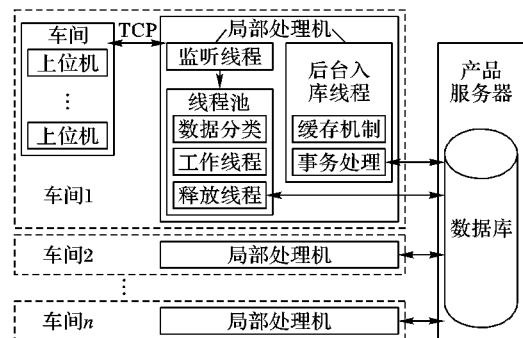


图4 系统模型示意图

局部处理机的设计模型如图 4 所示。从图中可知,局部处理机和车间的关系是一一对应的,而在一个车间里,上位机与处理机的比例为  $n:1$ ,各处理机将接收到的数据上传至统一的数据库中,故处理机与产品服务器的比例也是  $n:1$ 。处理机的监听线程采用服务器的方式进行循环监听,以保证工作时间都能实时接收数据。接收到的数据进行类别判断后,启动不同功能的工作线程进行计算处理,处理结果由入库线程上传至数据库中。为保证多路数据实时入库的准确性,工作线程与入库线程之间按生产者—消费者模式设计成前后台

关系,并配以缓存机制,以保障处理机与数据库之间相关事务均能及时准确地得到处理。

在实际生产中,处理机会产生许多的线程数,这也会带来一个问题,如果大量的上位机同时发出请求,中央处理器(Central Processing Unit, CPU)就要有很多开销用于创建、销毁线程,同时 Java 虚拟机(Java Virtual Machine, JVM)允许的最大线程个数并非没有限制,如果在多线程环境下,不受限制地建立线程会产生“Out Of Memory Error”异常<sup>[6]</sup>。为避免上述情况带来的工作异常<sup>[7-8]</sup>,处理机的工作线程采用线程池的方式开发,利用线程池的原理对线程资源进行管理、调度,保障系统健壮、平稳地运行<sup>[9]</sup>。

处理机的监听线程、工作线程、入库线程各司其职,负责实现不同阶段的计算任务,图5所示为1台处理机接收到1条数据时,其CPU时间片的分配说明图,从图中可见,数据处理是其中最费时的处理过程。图6所示为1台处理机采用多类线程并发的架构处理接收到的若干条数据时,其CPU时间片的分配说明图,需要说明的是,不同类别的数据其处理时长并不相同,由图6可见,处理机定义多类不同功能的线程可满足系统的实时要求,充分发挥CPU的并发特性。

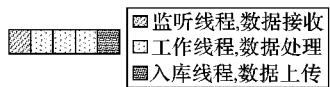


图5 1条数据处理的CPU时间片分配

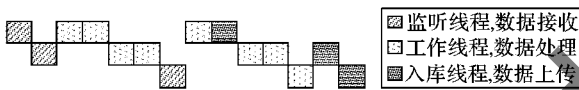


图6 3条数据并发处理的CPU时间片分配

## 2.2 关键技术与实现

### 2.2.1 设备工况类数据

设备类数据分为两类:实时信息、状态信息。实时信息计算处理后,立刻上传至数据库中供用户实时查看;状态信息计算处理后需暂存在本地处理机中,待接收到的若干条状态信息可组成一条完整的日志记录后,再上传至数据库中供用户查看。图7是设备工况类数据的处理流程,两类信息对应不同功能的工作线程及不同的入库接口。

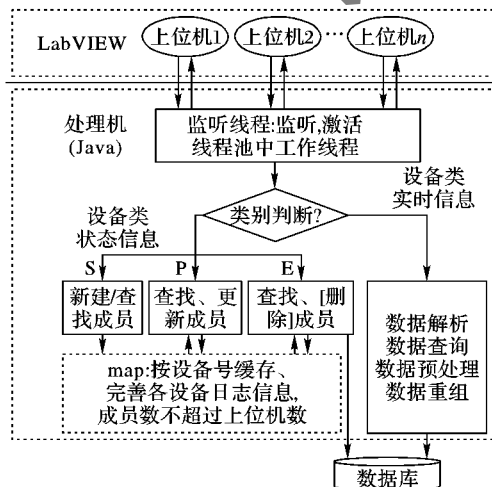


图7 设备工况类数据处理流程

监听线程接收到数据立即进行判断,如果是设备类的实时信息,则启动线程池中处理实时信息的工作线程,根据协议将数据解析,再根据用户的不同需求进行查询、预处理、重组

等计算,完成后即刻通知入库线程将结果数据实时传至数据库中。

如果是设备类的状态信息则启动线程池中处理状态信息的工作线程进行处理。由于单独的1条状态数据被看作是1个离散点,线程需要将若干离散数据先缓存,直到组织成一条完整的日志记录后,再通知入库线程传至数据库中。

状态信息分为3种:起始信息S、过程信息P、结束信息E。收到S类信息代表新建1条新的日志记录,收到P类信息代表更新、完善日志记录,收到E类信息代表1条日志记录已经完整,可以上传数据库。需要注意的是S、E两类信息有重叠部分,用R表示。为了保证数据能完整地缓存在本地,程序采用HashMap<String, MsgGK>结构来缓存各设备的日志记录,该结构命名为map。其中:String类型的设备编号作为map的索引Key;而MsgGK是自定义的设备日志类,该类根据协议将所有的离散数据进行了封装。接下来举例说明如何生成一条完整的日志。如果处理机接收到某设备的状态信息为m,工作线程首先对map进行检索,如果map中无此设备号,则说明m是S类,将该设备作为map的新成员添加进去;如果map中查到相同的设备号,则说明m是E类,将该设备在MsgGK中的现有信息根据协议整理成一条完整的日志,即可上传至数据库中,接着再将m归为S类,map中该设备的MsgGK信息中除设备编号以外的其他字段均置为初始值,m又开始一条新日志的“新建”状态。什么时候可以将map中的成员删除呢?只有当处理机接收到E类中非R的状态点,例如关机、故障等,则一方面表示该设备的日志记录已完整,可上传至数据库中;另一方面表示该设备已经终止与处理机的通信,可以从map中删除该成员了。以上的设计方式,一方面保证了map的最大成员数不超过上位机的总数,另一方面可保证map中同一设备成员在连续工作中的新建操作只执行一次。这两个因素使得处理机在本地缓存的日志数量始终控制在上位机总数以内,从而避免过度的内存开销或内存泄露<sup>[10]</sup>,有效减少计算时长,提高处理机的整体性能。

以下为实现工况类数据通信的相关程序的部分伪代码:

1)若干上位机的工况日志信息记录类GKMap的伪代码:

```
public class GKMap {
    //静态变量 map: 存储通信状态中的上位机的工况数据;
    //MsgGK 类是若干字段组成的工况日志信息类
    public static HashMap<String, MsgGK> map =
        new HashMap<String, MsgGK>();
    //成员方法: map 成员基本信息的更新方法 ResetMap
    public void ResetMap(String eqpid, MsgGK gkmsg) {
        根据 eqpid 找到 map 对应成员, 将 MsgGK 中基本信息置为初始状态;
        将新的工况数据 gkmsg 保存在对应成员数据中;
    }
}
```

2)管理工况信息读写的类 ManageGKThread 的伪代码,保障有序地对多个通道传送来的工况数据进行读写。

```
public class ManageGKThread {
    //成员变量: 工况信息 rec; 信息读取状态 isget
    private String rec;
    private int isget = 0; //0 表示不可读, 1 表示可读
    //成员方法: 读信息的方法 getRec
    public synchronized String getRec() {
```



则是对应某一台上位机。在 Java 的并发机制下,为了控制数据库的连接数以及并发事务数量,文档类数据的入库工作则由图 4 所示的后台入库线程提供接口统一实现。

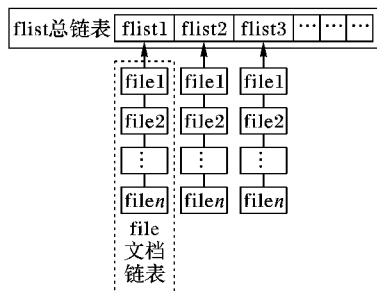


图8 “主从结构”双链表示意图

文档类工作线程和后台入库线程采用生产者—消费者模式<sup>[11]</sup>,flist 的链表为空时,入库线程为 wait 阻塞状态;链表不为空时,则唤醒入库线程,根据上位机 file 链表中的文件名依次上传对应文件中的数据。当一个 file 链表里记录的每份文件都上传完毕,则删除其对应的本地缓存文件,再从 flist 链表中删除该 file 成员;当 flist 中所有的 file 链表均处理完毕,即 flist 链表为空时,入库线程又进入 wait 阻塞状态。实现该模式时,flist 链表设计为文档类工作线程和入库线程的共享资源,在保证共享资源安全性<sup>[12]</sup>的同时,还要考虑程序执行的效率,故算法对 flist 的上传、删除操作均需对资源加锁,以保证与数据库通信时的数据有序性及正确性,而对 flist 的添加操作未加锁,以满足多台上位机能同步与处理机通信的需求。

以下为实现双链表结构管理文档类数据的相关程序的部分伪代码:

1)file 文档链表类 FilrPerThread 的伪代码:

```
public class FilrPerThread {
//成员变量:记录各文档名称的文档链表 filelst
private List<String> filelst = new ArrayList<String>();
//成员方法:添加新文档的方法 addfile
public void addfile(String filename) {
    filelst.add(filename);
}
//成员方法:获取文档链表的方法 getFilelst
public List<String> getFilelst() {
    return filelst;
}
//成员方法:清空文档链表的方法 cleafile
public void cleafile() {
    filelst.clear();
}
}
```

2)管理 flist 总链表读写的类 ManageThread 的伪代码,保障有序的将多个通道传送来的文档数据进行读写。

```
public class ManageThread {
//静态变量 recfile:即 flist 总链表,存储所有的 file 文档链表
public static List<FilrPerThread> recfile =
    new ArrayList<FilrPerThread>();
//静态变量 lstfull: flist 总链表的成员数,0 为空,1 为不为空
public static int lstfull = 0;
//静态方法 addfile:向 flist 总链表中添加新的 file 链表成员
public static void addfile(FilrPerThread onelst) {
    recfile.add(onelst);
}
}
```

//静态方法 notelstfull:如果 flist 总链表成员数不为空,唤醒工作  
//线程

```
public synchronized static void notelstfull() {
    ManageThread.lstfull = 1;
    ManageThread.class.notifyAll();
}
```

//静态方法 opefile:如果 flist 总链表成员数为空,工作线程  
//等待

```
public synchronized static void opefile() {
    while( lstfull == 0) {
        ManageThread.class.wait();
    }
    while( recfile.size() > 0) {
        //只要 flist 总链表成员数不为空,就循环
        FilrPerThread opelst = recfile.get(0);
        //取出一个成员 file 文档链表
        DBSerThread dbthread =
            new DBSerThread(opelst);
        dbthread.start();
        //工作线程启动
        //与文档接收线程 manageservermain 之间的交互工作;
        recfile.remove(opelst);
        //已处理过的 file 文档链表从 flist 总链表中删除
        opelst.cleafile();
        opelst = null;
        //删除已处理的 file 文档链表对象
    }
    //与文档接收线程 manageservermain 之间的交互工作;
    if( recfile.size() == 0) ManageThread.lstfull = 0;
}
```

3)文档类数据处理的工作线程 DBSerThread 伪代码:

```
public class DBSerThread extends Thread{
//成员变量:正在处理的 file 链表对象 opelst
FilrPerThread opelst = null;
//run 方法:
public void run() {
    List<String> filelst = opelst.getFilelst();
    // filelst 表示正在处理的 file 文档链表
    while( filelst 的未处理文档数不为零){
        MsgPenHead penhead =
            new MsgPenHead();
        //MsgPenHead 类是若干字段组成的文档头信息类
        penhead 按协议从某文档中获取相应数据;
        new DBCon(...).setmsgpenhead(penhead);
        //启动数据库线程,处理各类文档头信息
        文档头处理完毕且操作正确,继续按协议处理文档详情;
    }
    filelst = null;
    与文档接收线程之间进行通信;
}
```

综上所述,本文的局部处理机定义了 map 结构、flist 双链表结构分别实现对设备类、文档类数据的本地缓存;设计多种功能的线程、线程池并存的工作架构,实现对各类数据监控、接收、分类、计算、上传的并发管理,该系统的顺序图如图 9 所示。

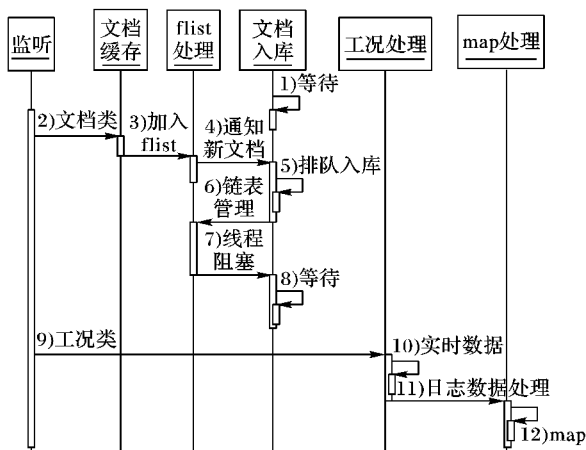


图9 系统程序顺序图

### 3 系统测试与分析

系统安装环境如图10所示。

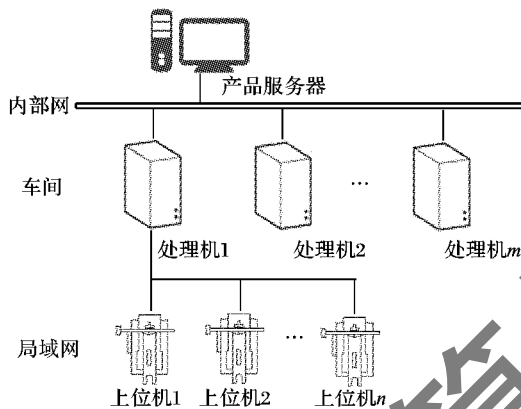


图10 安装环境示意图

图11所示为JConsole对程序的内存使用进行监测的结果示意图,横轴为时刻,纵轴为内存量,测试时间从10:40—02:50,在该时间段内,也有过一些极限测试(所有的上位机同时发送大文件、工况信息等)。由图11可知,没有产生内存泄露的情况,且各个上位机停止通信或间隔一定时间后,系统会回收不用的空间,内存使用能够有规律地回落,故通过图示可知处理机能正常运行。而分布式系统中的客户端能通过网页同步观测到各类文档数据、设备实时信息、设备日志信息。目前该系统已投入使用,并能正常准确地连续运行。

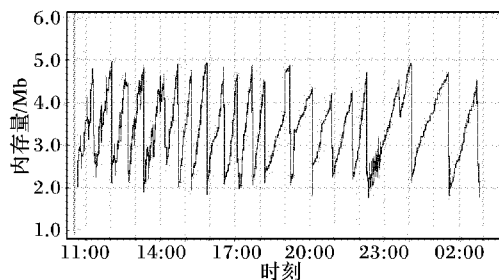


图11 内存使用情况示意图

### 4 结语

本文首先介绍了Hadoop的结构特征,了解到在并发(行)的工作模式中数据资源的管理、多线程的架构对于系统高效、准确地运行的重要性。在此基础上根据局部处理机实

际应用的工作环境以及客户需求,对处理机的多种工作线程互为关联的程序构架和配套的两种本地缓存方式进行了说明。目前该系统已投入使用,处理机的多线程并发结构兼顾到CPU和内存的工作特点,有效地保障了分布式系统整体的正常运行,通过一段时间连续工作表明,多线程的程序架构和有效的缓存方式能满足大量数据的同步处理及分析。

#### 参考文献:

- [1] ADYA A, HOWELL J, THEIMER M, *et al.* Cooperative task management without manual stack management[C]// Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference. Berkeley: USENIX Association, 2002: 289–302.
- [2] Apache Hadoop[EB/OL]. [2014-12-06]. <http://hadoop.apache.org>.
- [3] WEISS G. Multi-Agent systems: a modern approach to distributed artificial intelligence[M]. Cambridge: MIT Press, 1999: 121–165.
- [4] LUO Y, SHAO Z. Progress and prospects of standardization for Agent technology[J]. Computer Applications and Software, 2009, 26(3): 179–183. (罗勇军, 邵志清. Agent技术的标准化进度与前景[J]. 计算机应用与软件, 2009, 26(3): 179–183.)
- [5] SUN Z, YUAN Z, HUANG Z. The application of Hadoop on the data-intensive computing[C/OL]. [2014-06-20]. <http://www.docin.com/p-159855715.html&endPro=true>.
- [6] LINDHOLM T, YELLIN F, BRACHA G, *et al.* The Java virtual machine specification: Java SE 7[M]. ZHOU Z, XUE D, WU P, *et al.*, translated. Beijing: China Machine Press, 2014: 1–51. (LINDHOLM T, YELLIN F, BRACHA G. Java虚拟机规范: Java SE 7版[M]. 周志明, 薛笛, 吴璞渊, 等, 译. 北京: 机械工业出版社, 2014: 1–51.)
- [7] LIU X, ZENG B. A scheme design of using thread-pool to solve concurrent requests on server[J]. Modern Electronic Technique, 2011, 34(15): 141–143. (刘新强, 曾兵义. 用线程池解决服务器并发请求的方案设计[J]. 现代电子技术, 2011, 34(15): 141–143.)
- [8] ZHANG Y. Thread pool in the concurrent server[J]. Computer and Digital Engineering, 2012, 40(7): 153–156. (张垠波. 线程池技术在并发服务器中的应用[J]. 计算机与数字工程, 2012, 40(7): 153–156.)
- [9] ECKEL B. Thinking in Java[M]. HOU J, translated. Beijing: China Machine Press, 2002: 594–647. (ECKEL B. Java编程思想[M]. 侯捷, 译. 北京: 机械工业出版社, 2002: 594–647.)
- [10] LEI D, ZENG Q. Summary-based method to improve memory leak analysis[J]. Application Research of Computers, 2011, 28(11): 4315–4319. (雷达, 曾庆凯. 基于摘要的内存泄露分析方法改进[J]. 计算机应用研究, 2011, 28(11): 4315–4319.)
- [11] GOETZ B, PEIERLS T, BLOCH J, *et al.* Java concurrency in practice[M]. TONG Y, translated. Beijing: China Machine Press, 2012: 73–77. (GOETZ B, PEIERLS T, BLOCH J, 等. Java并发编程实践[M]. 童云兰, 译. 北京: 机械工业出版社, 2012: 73–77.)
- [12] ZHOU Z, ZHANG D, MIAO L. Model checking the concurrent programs based on the Java memory model[J]. Computer Engineering and Science, 2010, 32(3): 111–114. (周志远, 张大方, 缪力. 基于Java内存模型的并发程序模型检测[J]. 计算机工程与科学, 2010, 32(3): 111–114.)